Efficient Dynamic Pinning of Parallelized Applications by Reinforcement Learning with Applications

Georgios C. Chasparis[1(✉)], Michael Rossbory[1], and Vladimir Janjic[2]

[1] Software Competence Center Hagenberg GmbH, Softwarepark 21, 4232 Hagenberg, Austria
{georgios.chasparis,michael.rossbory}@scch.at

[2] School of Computer Science, University of St Andrews, Scotland, UK
vj32@st-andrews.ac.uk

Abstract. This paper describes a dynamic framework for mapping the threads of parallel applications to the computation cores of parallel systems. We propose a feedback-based mechanism where the performance of each thread is collected and used to drive the *reinforcement-learning* policy of assigning affinities of threads to CPU cores. The proposed framework is flexible enough to address different optimization criteria, such as maximum processing speed and minimum speed variance among threads. We evaluate the framework on the Ant Colony optimization parallel benchmark from the heuristic optimization application domain, and demonstrate that we can achieve an improvement of 12% in the execution time compared to the default operating system scheduling/mapping of threads under varying availability of resources (e.g. when multiple applications are running on the same system).

1 Introduction

Resource allocation is an indispensable part of the design of any engineering system that consumes resources, such as electricity power in home energy management [1], access bandwidth and battery life in wireless communications [10], computing bandwidth under certain QoS requirements [2] and computing bandwidth and memory in parallelized applications [4]. In this paper, we are focusing on the problem of allocating CPU cores to the tasks/threads of a parallel application (sometimes referred to as *mapping*). When resource allocation is performed online and the number, arrival and departure times of the tasks are not known a priori, the role of a *resource manager* is to guarantee the *efficient* operation (according to some criteria) of all tasks by appropriately allocating resources to them. This requires formulation of a centralized optimization problem (e.g., mixed-integer linear programming formulations [2]). However, it is usually difficult to formulate the problem precisely, and the methods to solve the resulting optimization problem are typically computationally very expensive. Additionally, most of the currently used allocation strategies [5,11,15] encounter issues

© Springer International Publishing AG 2017
F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 164–176, 2017.
DOI: 10.1007/978-3-319-64203-1_12

georgios.chasparis@scch.at

when dealing with *dynamic* environments (e.g., varying availability of resources), such as information complexity involved in retrieving the exact affinity relations during runtime and slow response to irregular application behaviour (e.g. degradation of performance due to presence of other applications). Such environments are suitable for *learning-based* optimization techniques, where the mapping/scheduling policy is updated based on performance measurements from the running threads. Through such learning-based scheme, we can (i) *reduce information complexity* when dealing with a large number of possible thread/memory bindings, since only performance measurements need to be collected during runtime; and, (ii) *adapt to uncertain/irregular application behavior*.

In our previous work [8], we have proposed a novel dynamic, *reinforcement-learning* based scheme for optimal allocation of parallel applications' threads to a set of available CPU cores. In this scheme, each thread responds to its current performance independently of other threads, requiring minimal information exchange. Furthermore, it exhibits robustness and is able to adapt to possible irregularities in the behavior of a thread (such as sudden drop of performance) or to possible changes in the availability of resources. In this paper, we extend the work presented there in two main directions:

- we introduce a new type of reinforcement-learning dynamics that allows faster adjustment towards better allocations;
- we evaluate the reinforcement-learning scheme on a real-world application (Ant Colony Optimization), demonstrating the reduction in application completion time of 12% compared to the default Linux Operating System scheduler.

These results are very encouraging, taking into account that our mechanisms does not require any input from the user.

The paper is organized as follows. Section 2 describes the overall framework and objective. Section 3 presents a reinforcement-learning algorithm for dynamic placement of threads. Section 4 presents experiments of the proposed algorithm in a Linux platform and comparison tests with the operating system's performance. Finally, Sect. 5 presents concluding remarks.

2 Problem Formulation and Objective

A substantial body of work has demonstrated the importance of the appropriate thread-to-core bindings in achieving a good performance of parallel applications. For example, Klug et al. [11] describe a tool that checks the performance of each of the available thread-to-core bindings and searches for an optimal placement. Unfortunately, this employs *exhaustive search*, which is usually prohibitively expensive. Broquedis et al. [5] combine the problem of thread scheduling with scheduling hints related to thread-memory affinity issues. These hints are able to accommodate load distribution given information for the application structure and the hardware topology. Scheduling itself is hierarchical, with *work stealing* [3] being used within neighboring cores to maintain data locality, while at

georgios.chasparis@scch.at

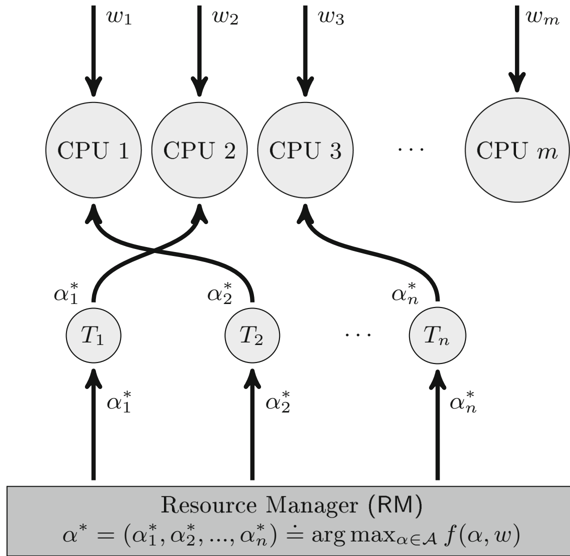

Fig. 1. Schematic of *static* resource allocation framework.

the memory-node level, the thread scheduler deals with larger groups of threads. A similar scheduling policy is also implemented by [14].

In this paper, we are interested in the problem of dynamic pinning of a set of threads $\mathcal{I} = \{1, 2, \ldots, n\}$ that comprise a parallel application to the set of (not necessarily homogeneous) CPU cores $\mathcal{J} = \{1, 2, \ldots, m\}$. We denote the *assignment* of a thread i to an available CPU by $\alpha_i \in \mathcal{A}_i \doteq \mathcal{J}$, i.e., α_i denotes the id of the CPU to which this thread has been assigned. Let also $\alpha = \{\alpha_i | i \in \mathcal{I}\}$ denote the *assignment profile*, which takes values on the Cartesian product $\mathcal{A} \doteq \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$. The *resource manager* (RM) periodically checks the performance of each thread and makes decisions about their pinning to CPUs so that a (user-specified) objective is maximized. Throughout the paper, we will assume that:

(i) The internal properties and details of the threads are not known to the resource manager. Instead, the resource manager may only have access to measurements related to their performance (e.g., their processing speed).
(ii) Threads may not be suspended and their execution cannot be postponed. Instead, the goal of the resource manager is to assign the *currently* available resources to the *currently* running threads.
(iii) Each thread may only be assigned to a single CPU core.

2.1 Static Optimization and Issues

Let $v_i = v_i(\alpha, w)$ denote the processing speed of thread i, which depends on both the overall assignment α, as well as external parameters aggregated within

georgios.chasparis@scch.at

w. The parameters w summarize, for example, the impact of other applications running on the same platform or other irregularities of the applications. The centralized objective for optimization is of the form

$$\max_{\alpha \in \mathcal{A}} f(\alpha, w). \tag{1}$$

In this paper, we will consider two different objectives, in order to show the flexibility of the proposed resource allocation scheme to address different optimization criteria. The considered objectives are the following:

(O1) $f(\alpha, w) \doteq \sum_{i=1}^{n} v_i/n$, corresponds to the *average processing speed of all threads*;
(O2) $f(\alpha, w) \doteq \sum_{i=1}^{n} [v_i - \gamma(v_i - \sum_{\ell=1}^{n} v_\ell/n)^2]/n$, for some $\gamma > 0$, corresponds to the *average processing speed minus a penalty that is proportional to the speed variance among threads*.

In the objective (O1), the goal is to minimize the average processing speed over all threads, and in the objective (O2) the goal is to achieve an optimal combination of processing speed and speed variance among threads.

Any solution to (1) corresponds to an *efficient assignment*. Figure 1 presents a schematic of a *static* resource allocation framework, where the centralized objective (1) is solved by the RM upfront, and then the optimal assignment (or mapping) is communicated to threads.

However, there are two significant issues when posing an optimization problem in the form of (1). In particular,

1. the function $v_i(\alpha, w)$ is *unknown* and it may only be approximated through measurements of the *processing speed*, denoted $\tilde{v}_i$;
2. the external influence w is *unknown* and may vary with time, thus the optimal assignment may not be fixed with time.

2.2 Measurement- or Learning-Based Optimization

We wish to target the objective (1) through a *measurement-based* (or *learning-based*) optimization approach. In such approach, the RM reacts to the approximation of the function $f(\alpha, w)$ that is obtained by measuring the processing speed of threads. Measurements are taken at time instances $k = 1, 2, \ldots$, and the approximation of function f at the time instance k is denoted by $\tilde{f}(k)$. For example, in the case of objective (O1), $\tilde{f}(k) \doteq \sum_{i=1}^{n} \tilde{v}_i(k)/n$. Given the approximation $\tilde{f}(k)$ and the current assignment of threads to cores, $\alpha(k)$, the RM selects the next assignment $\alpha(k+1)$ so that the measured objective approaches the true optimum of the unknown function $f(\alpha, w)$. In other words, the RM employs an update rule of the form:

$$\{(\tilde{v}_i(1), \alpha_i(1)), \ldots, (\tilde{v}_i(k), \alpha_i(k))\}_i \mapsto \{\alpha_i(k+1)\}_i \tag{2}$$

according to which prior pairs of measurements and assignments for each thread i are mapped into a new assignment $\alpha_i(k+1)$ that will be employed during the next evaluation interval.

georgios.chasparis@scch.at

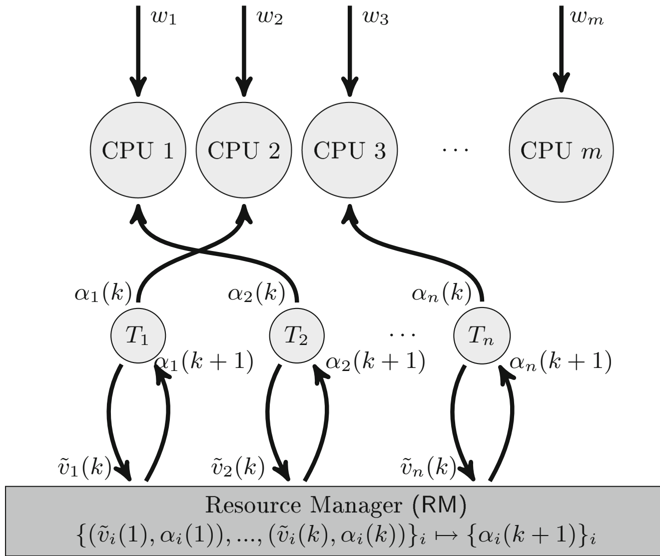

Fig. 2. Schematic of *dynamic* resource allocation framework.

The overall framework is illustrated in Fig. 2, describing the flow of information and steps executed. In particular, at any given time instance $k = 1, 2, \ldots$, each thread i communicates to the RM its current processing speed $\tilde{v}_i(k)$. Then the RM updates the assignment, $\alpha_i(k + 1)$, and communicates it to i.

2.3 Objective

The goal of our work is to utilize a distributed learning framework for *dynamic* (*adaptive*) pinning of threads to cores. Each thread constitutes an independent decision maker. It selects the CPU core to which it is pinned independently of others, using its own preference criterion. The job of the RM is to collect performance information and send it to the threads so that they can make the placement decisions. Our goal is to design a preference criterion and a selection rule for each thread so that maximizing the thread's own *criterion* ensures certain overall performance for the parallel application. Furthermore, the selection criterion of each thread should be adaptive and robust to possible resource variations. In the next section, we present such a (distributed) learning scheme.

3 Reinforcement Learning (RL)

The question that naturally emerges is *how should threads choose CPU cores based only on their available measurements, so that eventually an efficient assignment is established for all threads.* We achieve this by using a learning framework, *perturbed learning automata*, that is based on the reinforcement learning

georgios.chasparis@scch.at

algorithm introduced by the authors in [6,7]. It belongs to the general class of *learning automata* [13]. The basic idea behind reinforcement learning is rather simple. Each *agent* i (in this case, a thread), keeps track of a strategy vector that holds its estimates over the best choice (in this case, the CPU core). We denote this strategy by $\sigma_i = [\sigma_{ij}]_{j \in \mathcal{A}_i} \in \Delta(|\mathcal{A}_i|)$, where $\Delta(m)$ denotes the *probability simplex* of size m, i.e., the set of probability vectors in $\mathbb{R}^m$. To provide an example, consider the case of 3 available CPU cores, i.e., $\mathcal{A}_i \equiv \mathcal{J} = \{1, 2, 3\}$. In this case, the strategy $\sigma_i \in \Delta(3)$ of thread i may take the following form:

$$\sigma_i = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.3 \end{pmatrix},$$

which denotes that there is 20% probability of assigning the thread i to the CPU core 1, 50% probability of assigning the thread i to the CPU core 2 and 30% probability of assigning the thread i to the core 3. We will denote the assignment selection by $\alpha_i = \mathsf{rand}_{\sigma_i}[\mathcal{A}_i]$.

Note that if σ_i is a unit vector e_j, with 0 in all places except for the j-th, and 1 in the j-th place, then the thread i will be mapped to the core j with probability one. Such a strategy is usually called *pure strategy*.

3.1 Strategy Update

According to the *perturbed reinforcement learning* [6,7], the probability that a thread i selects action j at time $k = 1, 2, \ldots$ is:

$$\sigma_{ij}(k) = (1 - \lambda)x_{ij}(k) - \frac{\lambda}{|\mathcal{A}_i|} \tag{3}$$

where $\lambda > 0$ corresponds to a perturbation term (or *mutation*) and $x_i = [x_{ij}]_j$ corresponds to the *nominal strategy* of thread i. The nominal strategy is updated according to the following recursion formula:

$$x_i(k+1) = \begin{cases} x_i(k) + \epsilon \cdot u_i(\alpha(k)) \cdot [e_{\alpha_i(k)} - x_i(k)], & u_i(\alpha(k)) > \bar{u}_i(k) \\ x_i(k), & u_i(\alpha(k)) \leq \bar{u}_i(k), \end{cases} \tag{4}$$

for some constant step size $\epsilon > 0$, where $\bar{u}_i(k)$ denotes the running-average performance at time k and $u_i(\alpha(k))$ is the *utility* of thread i at time k, defined as $u_i(\alpha(k)) = \tilde{f}(k)$. In other words, each thread is assigned a performance index that coincides with the overall objective function (*identical interest*). In words, according to (4), if the performance of thread i at time k, when placed on core $\alpha_i(k)$, is higher than the average performance, i.e., $u_i(\alpha(k)) > \bar{u}_i(k)$, then at time $k + 1$ we increase the probability of that thread being placed on the same core and proportionally to the thread utilisation. So the better the thread performs, the more likely it is to be assigned to the same core. Otherwise, if the performance of the thread is the same or worse than the average, we do not change preference for its placement (the second case in (4)). In comparison to

georgios.chasparis@scch.at

our previous work [6,7], here we use the constant step size $\epsilon > 0$ (instead of a decreasing step-size sequence). This increases the adaptivity and robustness of the algorithm to possible changes in the environment. This is because a constant step size provides a fast transition of the nominal strategy from one pure strategy to another. Compared to [8], here we use a different reinforcement direction. In the Eq. (4), the strategy vector is only adjusted when a performance is higher than the running-average performance $\bar{u}_i$, which provides a faster adjustment towards better assignments. The perturbation term λ provides the possibility for the nominal strategy to escape (suboptimal) pure strategy profiles. Setting $\lambda > 0$ is essential for providing an adaptive response of the algorithm to changes in the environment.

The convergence properties of this class of dynamics can be derived following the exact same reasoning used for the learning dynamics presented in [8]. In fact, it can be shown that the dynamics approach asymptotically a set of allocations that includes the solutions of the centralized optimization (1). Such a set may in fact include sub-optimal allocations; however, as we shall see in the forthcoming evaluation section, they are still notably better that the allocations provided by the default operating system scheduler.

As a final notice, the algorithm is augmented with a *reset* strategy when a thread becomes inactive (e.g., due to termination), in which case the assignment profile is reset based on a round-robin initialization strategy.

3.2 Discussion

The reinforcement-learning algorithm of Eq. (4) provides a performance-based optimization. No a-priori knowledge of the type of the application or the underlying hardware is necessary. Furthermore, its memory complexity is minimal, since at any update instance of the resource manager, only the strategy vectors of each one of the threads needs to be kept in memory, whose size is linear to the number of the CPU cores. Furthermore, for each thread, the dynamics exhibit linear complexity to the number of CPU cores.

4 Experiments

In this section, we present an experimental study of the proposed reinforcement learning scheme for dynamic pinning of threads of parallel applications. The experiments were conducted on `20×Intel©Xeon©CPU E5-2650 v3 2.30 GHz` running Linux Kernel 64bit 3.13.0-43-generic. The machine divides the physical cores into two NUMA nodes (Node 1: CPUs 0–9, Node 2: CPUs 10–19). As an example application, we consider a parallel implementation of the Ant Colony Optimization heuristic for solving NP-complete optimization problems. The proposed reinforcement learning dynamics is implemented in scenarios under which the availability of resources may vary with time. We compare the overall performance of the algorithm, with respect to the completion time of the application.

georgios.chasparis@scch.at

4.1 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) [9] is an optimization algorithm used for solving NP-hard combinatorial optimization problems. The metaheuristics, given in Algorithm 1, consist of a number of iterations. In each iteration, each individual agent (*ant*) independently finds a solution to a given problem. The solution is biased by a pheromone trail (t), which is stronger along previously successful routes. After all ants have computed their solution, the best solution is chosen and, if needed, the pheromone trail is updated according to the quality of the new best solution. After that, the next iteration starts. The metaheuristics are applied to a specific problem by providing the objective function, evaluate the solution and update the pheromone trail.

```
Data: Ants - a set of ants
p - a set of problem parameters
t - pheromone trail
Result: best_result
initialization;
for i = 0 to i < num_iter do
    foreach a ∈ Ants do
        a = find_one_solution(p,t);
    end
    best=choose_best_solution(Ants);
    t = update_pheromone_trail(best, t);
end
```

Algorithm 1. Pseudocode of metaheuristics in ACO.

In this paper, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP). We are given n jobs. Each job, i, is characterised by its processing time, p_i (p in the code below), deadline, d_i (d in the code below), and weight, w_i (w in the code below). The goal is to find the schedule of jobs that minimizes the total weighted *tardiness*, defined as $\sum w_i \cdot \max\{0, C_i - d_i\}$ where C_i is the completion time of the job, i. The pheromone trail is defined as a matrix τ, where $\tau[i, j]$ is a real number between 0 and 1 that represents preference of putting job i at the j-th place in the schedule. The pseudocode for a function to find one solution is given in Algorithm 2. It iterates over the positions in the schedule. For each position, first an auxiliary function ϵ is applied for each job to compute the probability of that job being assigned to that position. This probability is then further tuned to take into account the pheromone trail τ. Then, according to some probability, one of the two actions are taken - either the job with the highest probability or a random job (according to the calculated probabilities). The latter is done to add a degree of randomisation to the solutions, in order to escape possible local maxima.

georgios.chasparis@scch.at

```
Data: p - a set of problem parameters
τ - initial pheromone trail
Result: schedule
for k = 0 to num_jobs do
    foreach unscheduled job i do
        // probability of selecting job i as the k-th in the schedule
        prob[i] = ε(i, p)^β · τ[k, i];
    end
    q = rand();
    if q < Q then
        job = select the job with the highest probability, according to prob;
    else
        job = select a random job, according to probabilities in prob;
    end
    schedule[k] = job;
end
```

Algorithm 2. Pseudocode for find_one_solution function for SMTWTP instance of ACO.

4.2 Parallelization and Experimental Setup

The ACO metaheuristics can be parallelized by dividing ants into groups and computing the find_one_solution function in Algorithm 1 for groups of ants in parallel. We consider a uniform division of ants to threads (*task farm* parallel pattern). Parallelization is performed using the `pthreads` parallel library.

Throughout the execution, and with a fixed period of 0.2 s, the RM collects measurements of the total instructions per sec (using the PAPI profiling library [12]) for each of the threads separately. Taking into account these measurements, the update rule of Eq. (4) under (O2) is executed by the RM. Pinning of the threads to the available CPUs is achieved with the `sched.h` library (in particular, the `pthread_setaffinity_np` function). In the following, we evaluate the completion time of the test application under the reinforcement-learning scheme, compared to the time achieved under the Linux Operating System (OS) default scheduling mechanism. We compare them for different values of $\gamma \geq 0$ in order to investigate the influence of more balanced speeds to the overall running time.

In all the forthcoming experiments, the RM is executed by the master thread which is always running on CPU 0. Furthermore, in all experiments, only the first one of the two NUMA nodes are utilized, since our intention is to investigate the benefit of efficient placement of thread to cores without taking into account effects of non-uniform memory layout on the execution speed.

4.3 Experiment 1: ACO Under Uniform CPU Availability

In the first experiment, we consider the ACO application consisting of 20 threads and utilizing 7 CPU cores. Table 1 shows the completion times under the OS

georgios.chasparis@scch.at

and reinforcement-learning (RL) for different values of $\gamma > 0$, with $\epsilon = 0.01$ and $\lambda = 0.03$ in formulas (3) and (4). We select a step size and perturbation that are not so small in order to allow a rather fast adaptation (via $\epsilon > 0$) and a rather often experimentation (via $\lambda > 0$).

Table 1. Statistical results regarding the completion time (in sec) of OS and RL under Experiment 1.

		$\epsilon = 0, \lambda = 0$	$\epsilon = 0.01$, $\lambda = 0.03$		
Run #	OS	RL ($\gamma = 0$)	RL ($\gamma = 0$)	RL ($\gamma = 0.02$)	RL ($\gamma = 0.04$)
1	138.39	139.41	142.08	142.69	141.69
2	138.57	137.60	143.28	141.69	141.27
3	138.80	138.39	142.87	142.10	140.92
4	138.38	137.97	144.08	143.47	142.71
5	138.78	138.40	143.28	142.65	141.28
Aver.	**138.58**	**138.35**	**143.12**	**142.52**	**141.57**
s.d.	**0.20**	**0.67**	**0.73**	**0.68**	**0.69**

We observe that the RL scheduler can almost match the completion time by the OS scheduler. The RL scheduler with $\gamma = 0.04$ gives just about 2.12% worse completion time, compared to the OS scheduler. This difference can be attributed to the necessary adaptation and experimentation incorporated into the scheduler. To see this, note that when the scheduler sticks with the initial round-robin static initialization of the assignments, i.e., when $\epsilon = \lambda = 0$, then the completion time matches very accurately the time achieved by the OS scheduler (Table 1). Such experimentation is absolutely necessary for the dynamic scheduler to be able to react to variations in the availability of resources, as it will become obvious in the following experiments.

Another interesting observation comes from the fact that as γ increases, the overall completion time of the application decreases. In other words, when penalizing high speed variance among threads, the overall completion time decreases. Such conclusion may not necessarily be generalized beyond this experimental setup of identical threads and uniform resource availability; however, it indicates a potential benefit that needs to be further investigated.

4.4 Experiment 2: ACO Under Non-uniform CPU Availability

In the second experiment, the execution speed of the CPU cores is not uniform. To achieve this variation, we have another (*exogenous*) application running on some of the available CPU cores. In particular, this exogenous application places equal work-load to the first three CPU cores. The exogenous application already runs when the ACO starts running. Figure 3 shows the running average processing speed under OS and RL, which is further supported by the statistical data of

georgios.chasparis@scch.at

Table 2. The RL achieves a significant speed improvement that results in about 12% reduction in completion time.

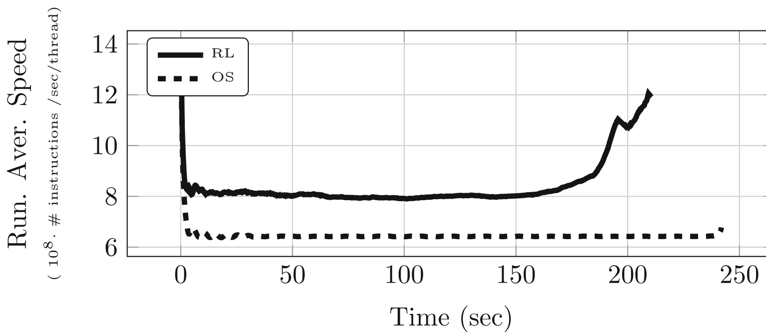

Fig. 3. Running average execution speed for OS and RL ($\gamma = 0.04$) under Experiment 2.

4.5 Experiment 3: ACO Under Time-Varying CPU Availability

This is an identical experiment to Experiment 2, except for the fact that the exogenous application starts running 30 s after ACO starts running. This form of test examines the ability of RL to respond after a significant variation in the availability of some of the CPU cores. Figure 4 illustrates the evolution of the running-average processing speed under OS and RL for this experiment.

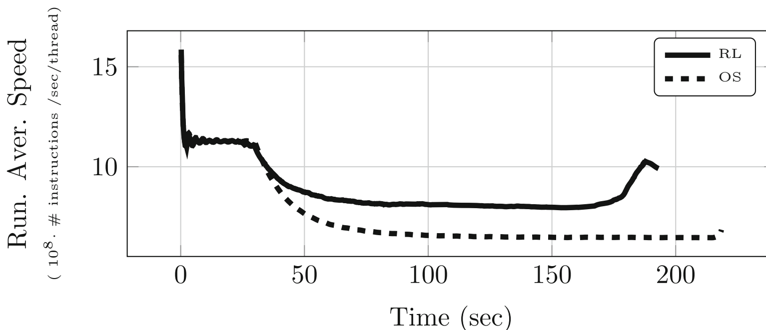

Fig. 4. Running average execution speed for OS and RL ($\gamma = 0.04$) under Experiment 3.

It is evident in Fig. 4 that the RL dynamic scheduler is able to better react to variations in the availability of resources, and achieves a shorter completion time by about 10%. This is also supported by the statistical data of Table 2.

georgios.chasparis@scch.at

Table 2. Statistical results of the completion time (in sec) under OS and RL in Experiments 2 and 3, respectively.

	Experiment 2		Experiment 3	
Run #	OS	RL ($\gamma = 0.04$)	OS	RL ($\gamma = 0.04$)
1	241.30	207.33	218.48	193.30
2	239.10	201.92	218.70	196.45
3	240.90	220.11	218.88	201.92
4	241.11	221.54	219.27	195.88
5	241.51	210.09	218.52	193.41
Aver.	**241.06**	**212.20**	**218.77**	**196.19**
s.d.	**0.99**	**8.42**	**0.33**	**3.50**

5 Conclusions and Future Work

We proposed a measurement-based reinforcement learning scheme for addressing the problem of efficient dynamic pinning of threads of a parallel application to the processing units. According to this scheme, a centralized objective is decomposed into thread-based objectives, where each thread is assigned its own utility function. A resource manager updates a strategy for each of the threads corresponding to its beliefs over the most beneficial CPU placement for this thread. Updates are based on a reinforcement learning rule, where prior actions are reinforced proportionally to the resulting utility. Besides its reduced computational complexity, the proposed scheme is adaptive and robust to possible changes in the environment. We further demonstrated that in the ACO metaheuristics algorithm, the proposed scheduler may reduce the completion time up to 12% under varying resource availability. This is a significant result, as the reinforcement-learning based scheduler does not require any input from the user, nor it requires any information from the application itself, therefore it can be readily plugged in instead of the default operating system scheduler. In future, we plan to investigate the effect of non-uniform memory layout to our scheduler and to adapt the scheduling policies for these kind of systems.

Acknowledgments. This work has been partially supported by the European Union grant EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

1. De Angelis, F., Boaro, M., Fuselli, D., Squartini, S., Piazza, F., Wei, Q.: Optimal home energy management under dynamic electrical and thermal constraints. IEEE Trans. Ind. Inform. **9**(3), 1518–1527 (2013). doi:10.1109/TII.2012.2230637. ISSN 1551-3203
2. Bini, E., Buttazzo, G.C., Eker, J., Schorr, S., Guerra, R., Fohler, G., Årzén, K.E., Vanessa, R., Scordino, C.: Resource management on multicore systems: the ACTORS approach. IEEE Micro **31**(3), 72–81 (2011)

georgios.chasparis@scch.at

3. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proceedings of SFCS 1994, pp. 356–368 (1994)
4. Brecht, T.: On the importance of parallel application placement in NUMA multiprocessors. In: Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), San Deigo, CA, pp. 1–18, July 1993
5. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.A., Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. Int. J. Parallel Program. **38**, 418–439 (2010)
6. Chasparis, G.C., Shamma, J.S., Rantzer, A.: Nonconvergence to saddle boundary points under perturbed reinforcement learning. Int. J. Game Theory **44**(3), 667–699 (2015)
7. Chasparis, G., Shamma, J.: Distributed dynamic reinforcement of efficient outcomes in multiagent coordination and network formation. Dyn. Games Appl. **2**(1), 18–50 (2012)
8. Chasparis, G.C., Rossbory, M.: Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning. arXiv:1606.08156 [cs], June 2016
9. Dorigo, M., Stützle, T.: Ant Colony Optimization. Bradford Company, Scituate (2004)
10. Inaltekin, H., Wicker, S.: A one-shot random access game for wireless networks. In: International Conference on Wireless Networks, Communications and Mobile Computing (2005)
11. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: `autopin` - automated optimization of thread-to-core pinning on multicore systems. In: Stenstrom, P. (ed.) Transactions on High-Performance Embedded Architectures and Compilers III. LNCS, vol. 6590, pp. 219–235. Springer, Berlin Heidelberg (2011). doi:10.1007/978-3-642-19448-1_12
12. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: A portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference, pp. 7–10 (1999)
13. Narendra, K., Thathachar, M.: Learning Automata: An introduction. Prentice-Hall, Upper Saddle River (1989)
14. Olivier, S., Porterfield, A., Wheeler, K.: Scheduling task parallelism on multi-socket multicore systems. In: ROSS 2011, Tuscon, Arizona, USA, pp. 49–56 (2011)
15. Thibault, S., Namyst, R., Wacrenier, P.-A.: Building portable thread schedulers for hierarchical multiprocessors: the bubblesched framework. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 42–51. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74466-5_6

georgios.chasparis@scch.at